

PROGRAMMING
EXPERT

FAQ

Q What's an IDE?

A Integrated development environments, or IDEs, make programming easier. In theory, you can write programs in a text editor and run them with an interpreter or compiler. The IDE integrates both steps and, crucially, understands your programming language. It can auto-complete commands, prompt for parameters and flag up errors. It also formats code to make it easier to understand.

Most IDEs can run programs in a controlled way that makes finding bugs easier. For example, you can stop the program and examine the contents of variables. An IDE helps manage projects made up of multiple programs and resource files.

Working without an IDE is now rare. Visual Studio 2005 is Microsoft's IDE for all the .NET languages. Other languages, such as Java, have several IDEs.

Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk

Easy thumbnails

Creating tiny image previews is easy with the Send To menu. **Mike James** shows how to convert an image into a thumbnail, customise its size and tweak your thumbnail utility



Thumbnails – small previews of full-sized images – are a great way of allowing users to select images to view or work with at full resolution. Creating a thumbnail from a high-resolution original is a task that programmers must often tackle, and one you'd expect to be simple. In reality, however, creating thumbnails has some interesting twists.

This month's project tackles the conversion of an image into a JPEG-compressed thumbnail. We'll also build an interface that enables the user to choose the thumbnail size. Finally, we'll integrate our thumbnail creator with Windows' Send To menu. The Send To menu appears when you right-click a file, giving you a choice of what to do with it. Adding items to the Send To menu is easy, and provides a great way of adding file-processing utilities to Windows. Once you know how it works, it's difficult not to invent new uses for it.

RETURN TO SEND TO

So how does Send To work? The answer, when you know how complex other Windows features are, is pleasingly simple. For each user, there is a special directory called:

Documents and Settings\username\SendTo

Any shortcuts that you place in this directory automatically appear in the Send To menu as options. It's that simple. You can place special items in the Send To directory. Drop Targets and DeskLink files, for instance, implement the My Documents and Desktop (create shortcut) items on the menu. However, these special items are just more subtle ways of getting applications running; there isn't a particular reason to consider how they work when a simple shortcut does the same thing. When a user selects some files, right-clicks and chooses a program from the Send To menu, the program starts with the full pathnames of the selected files as a set of command-line arguments.

For a quick demo, find Notepad in the Start menu and use Ctrl-C to copy this shortcut to the clipboard. Next, navigate to the Send To directory (the only problem you might have is working out the correct

username) and use Ctrl-V to paste the shortcut there. Now when you right-click any file and hover over Send To, you'll see Notepad as a new option. If you select this option, Notepad opens with the file ready to edit.

INSTALLING A SEND TO UTILITY

We could ask the user to create a shortcut for our thumbnail utility in the Send To directory. However, it's not too difficult to do this automatically. First, start a new C# Windows project called Thumber. To work as a Send To item, this needs to access the command-line arguments it's passed. In last month's project, we showed you how to do this by changing the main function and the form's constructor. In fact, there is an easier way to get the command-line arguments using the Environment class. In the Form's constructor add:

```
public Form1()
{
    InitializeComponent();
    m_args = Environment.GetCommandLineArgs();
}
```

We also need a global variable to store the arguments:

```
private string[] m_args;
```



▲ Adding a new Send To menu item is very easy

The only real difference between the methods of retrieving the command-line arguments is that the Environment class also returns the first element of the array containing the name of the executable program. Unfortunately, some versions of Windows return this with a full pathname and others return just the name of the executable, which makes it more difficult to use.

We can assume that if the program is run with no additional command-line arguments, it hasn't been started with the Send To command; if it had, there would be the name of at least one file passed to it. If it is run in this way, we can set things up so that it installs itself into the Send To directory – that is, running the application directly installs it. You could extend the code shown here to display a pop-up dialog box that asks the user if they really want to install it, or gives installation and uninstallation options.

We need the location of the Send To directory for the current user and the current directory in which the executable is stored. Both are easy to get, again with the help of the Environment class:

```
if (m_args.Length == 1)
{
    string SendToDir =
        Environment.GetFolderPath(
            Environment.SpecialFolder.
            SendTo);
    string ProgramPath =
        Environment.CurrentDirectory;
```

You can get the location of any special folder using the same class.

We're ready to create the shortcut, but there is a problem. Although the .NET Framework has lots of helpful classes, there doesn't seem to be one that lets you work with shortcuts. This has several solutions, including using `plnvoke` to call low-level API functions. The simplest solution, however, is to use an ActiveX class library that is available as part of the Scripting support (for both JScript and VBScript). To use it, first add a reference to the Windows Script Host Object Model; you'll find it on the COM tab. Next add:

```
using IWshRuntimeLibrary;
```

Now we can create a shortcut in exactly the same way we would using VBScript or JScript. First create a `WshShell` object:

```
WshShellClass WshS = new
    WshShellClass();
```

Then use the object to create a shortcut object:

```
IWshShortcut Shortcut= (IWshShortcut)
    WshS.CreateShortcut(SendToDir+"\\
    Thumber.lnk");
```

We have to specify the name and location of the new shortcut. Then we customise the shortcut object using its properties:

```
WshShellClass WshS = new
    WshShellClass();
IWshShortcut Shortcut=
    (IWshShortcut) WshS.CreateShortcut
    (SendToDir+"\\Thumber.lnk");
Shortcut.TargetPath =
    @"\\thumber.exe";
Shortcut.WindowStyle = 1;
```

There are other properties that you can set, such as a custom icon, but these are all we need to get our program working. Finally, we save the Shortcut object, and this creates the Thumber.lnk shortcut file:

```
Shortcut.Save();
}
```

If the lnk file already exists, it is deleted and rewritten. There will probably be an easier way to create a shortcut in a future version of the .NET Framework; if so, use it.

THE THUMBNAIL INTERFACE

Now that we've finished with the installation of the Send To option, we need to consider our user interface (UI). The idea is to show the user a thumbnail version of one of the files and let them drag a slider to change its size. When they have selected a size, clicking on a button creates thumbnails for all the files.

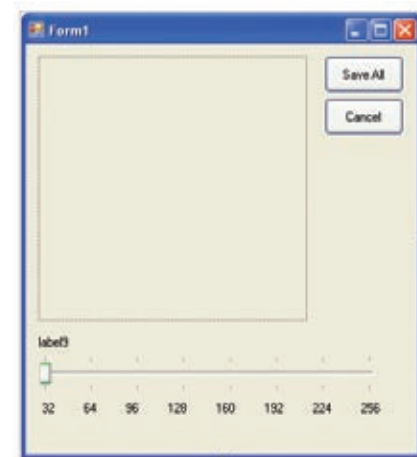
Place a PictureBox on the form in the top left-hand corner and use the Properties window to set its size to 256, 256 and its size mode to Autosize. Place a TrackBar below the PictureBox, and use the Properties window to set its LargeChange, SmallChange, TickFrequency and Minimum properties to 32. Set the Maximum property to 256. Finally, place labels below the ticks on the TrackBar, a further label (called label9) just below the PictureBox, and two buttons called SaveAll and Cancel on the form. The final layout should look like the screen in the picture below.

GETTING THE PICTURE

When the application opens with more than just the executable name in the arguments, there are some picture files to process. We need to read in one of the files, which might as well be the first in the list, and remember the path of the directory in which the files are stored:

```
if (m_args.Length > 1)
{
    m_first= getImage(m_args[1]);
    m_dir = system.io.Path.GetDirectory
        Name(m_args[1]);
}
```

The Path class (from the system.IO namespace) provides useful filename-processing methods. You need never use general string manipulation to extract a filename or directory because the Path class does everything you want. In this case,



▲ The user interface for selecting a thumbnail size

it simply strips out the directory part of the path, removing the filename.

We need to add two global variables:

```
private Bitmap m_first;
private string m_dir;
```

To use the graphics classes easily, we also need to add (to the start of the project):

```
using System.Drawing.Imaging;
```

We then create a `getImage` function that reads a file into a Bitmap class:

```
public Bitmap getImage(string name)
{
    return new Bitmap(name);
}
```

For simplicity, we've omitted to check that 'name' specifies a valid graphics file. You should add such a check if you want the program to be robust.

Finally, we call the TrackBar's Scroll event handler, as this contains the code to load and process the first picture:

```
trackBar1.Value = 256;
trackBar1.Scroll(trackBar1, new
    EventArgs());
}
```

The TrackBar's value is initially set to 256 to display the largest possible thumbnail. This completes the form's constructor, the routine starting with 'public Form1()':

THE EVENT HANDLER

The Scroll event is raised every time the user moves the TrackBar's slider. It responds by redrawing the thumbnail at the size selected. As the first file is already stored in `m_first`, we can use this to discover the picture's current size. When you make a thumbnail of the image, you can choose to keep the height-to-width ratio the same as the original, or to squash or crop it to fit into a square.

In our case, we'll use the TrackBar's value to set the height of the image, and the width will be calculated in proportion to the original:

```
private void trackBar1_Scroll(
    object sender, EventArgs e)
{
    m_h = trackBar1.Value;
    m_w = m_h * m_first.Width /
        m_first.Height ;
```

As `m_w` and `m_h` are going to be used to scale all the images when the user clicks the appropriate button, they need to be global variables:

```
private int m_w, m_h;
```

We'll use `label9` to let the user know the size of the thumbnail selected:

```
label9.Text = m_w.ToString() +
    " X " + m_h.ToString();
```

Finally, we need to create the thumbnail and display it in the PictureBox:

NO STARCH PRESS

The Book of JavaScript:
A Practical Guide to
Interactive Web Pages

★★★★

£18

JavaScript is a hot topic. With Microsoft opting for JScript as its main scripting language, and with JavaScript-based AJAX becoming ever more popular, JavaScript is almost the only scripting language worth using. As a result, there are lots of new and revamped JavaScript books hitting the shelves. Dave Thau's book is one of the many volumes hoping to cash in.

Far from introducing JavaScript in a sensible order to suit beginners, it's a disorganised mess. Its main failing is that it fails to distinguish between JavaScript and related technologies such as DHTML, CSS and the DOM. Its task-oriented approach is good, though, with examples that show how to implement rollovers, menus and so on. It has new chapters on Ajax, but they are uninspiring and fail to explain the ideas in a clear or motivating way.

If you need a mixed-up, traditional view of JavaScript with lots of standard examples, you may find this useful, but there are better choices.

SUMMARY

- ✓ Task-oriented, with examples
- ✗ Too disorganised for beginners

SUPPLIER www.amazon.co.uk
ISBN 1593271069
PAGES 490



NEW RIDERS

Codin' for the Web

★★★★★

£19

Website creators can only get so far without programming. Once you realise that you need more than just HTML, Charles Wyke-Smith's latest book provides a gentle introduction to programming webpages using PHP.

Learning to program from scratch is tough, and the complexities of HTML and the distinction between client-side and server-side actions make it all the more difficult. Under the circumstances, the author does a remarkably good job. He understands the sorts of things that are likely to trouble the beginner and does his best to introduce ideas in a logical order, complete with notes to make sure you understand. It's not perfect, because the task is a difficult one, but the examples are simple, and it's well produced with full colour throughout.

Being a server-side language, PHP is more powerful than client-side JavaScript but is also potentially more confusing. If you are prepared to put the effort into following the explanations and the examples in this book, you'll make a good start on the road to being a web programmer.

SUMMARY

- ✓ Well explained, with examples
- ✗ PHP is hard for beginners

SUPPLIER www.amazon.co.uk
ISBN 0321429192
PAGES 286



```
pictureBox1.Image = thumb(m_first,
    m_w, m_h);
}

```

We've separated the graphics work into a different function called Thumb. Fortunately, writing the thumb function is easy:

```
public Bitmap thumb(Bitmap image,
    int m, int n)
{
    Bitmap temp =
        (Bitmap)image.GetThumbnailImage(
            m, n,
            null, IntPtr.Zero);
    return temp;
}

```

This uses the GetThumbnailImage method, which every Bitmap object has.

If the image has a built-in thumbnail, it's this thumbnail that is resized. This could be a problem, but it certainly isn't with JPEGs and TIFFs. If it is a problem, however, you can use one of the more general Bitmap redrawing methods.

CONVERTING THE LIST

At this point we have an application that loads one of the specified pictures and enables the user to view it as a thumbnail and control its size with a slider. When the user clicks on the Save All button, we have to process all the other images listed in the command-line arguments. First, we need to decide where to store the thumbnails. We could store them back in the same directory with altered filenames. Our preferred solution is to create a Thumbs sub-directory and store the thumbnails there, using the same name as the file from which each is derived. Creating a directory is easy, because we already have the pathname of the directory in which the pictures are stored:

```
private void button1_Click(

```

```
    object sender, EventArgs e)
{
    string thumbdir = m_dir +
        @"\Thumbs\";
    Directory.CreateDirectory(
        thumbdir);
}

```

The next task may seem to be out of sequence, but we have to get ready to save the thumbnails in a suitable compressed format. You might think that saving a bitmap as a JPEG is such a common job that it must be easy. It has been made less tricky in .NET 3.0 (still in beta), but it's still not easy – not much easier than the .NET 2.0 solution presented here.

To make a JPEG in .NET 2.0, you get a suitable encoder object, customise it using an EncoderParameters object and use it as part of the Bitmap Save method. The first problem is getting a suitable encoder. Amazingly, there is no method that does this in the .NET Framework. It isn't difficult to make your own, but it shouldn't be necessary. We need to write a function, GetEncoderInfo, that will find the appropriate Encoder for the image format we want to use. The GetImageEncoders method returns an array of ImageCodecInfo objects, one for each available encoder. Once we have this, it's easy to scan for the encoder we want:

```
public static ImageCodecInfo
GetEncoderInfo(ImageFormat type)
{
    ImageCodecInfo[] encoders;
    encoders =
        ImageCodecInfo.GetImageEncoders();
    foreach(ImageCodecInfo encoder in
        encoders)
    {
        if (encoder.FormatID.Equals
            (type.Guid))
            return encoder;
    }
    return null;
}

```

The GetEncoderInfo function returns an ImageCodecInfo object that represents the encoder that matches the graphics type you specify – in this case, ImageFormat.Jpeg. You can search for encoders based on the file extensions they handle, the encoder description or the MIMETYPE, but using the ImageFormat seems to be the most direct way. The function returns a null if it can't find an encoder of the correct type.

Once we have this function, we can return to finish the Button click event handler and get the encoder we need:

```
ImageCodecInfo jpegCodec =
    GetEncoderInfo(ImageFormat.Jpeg);

```

To control the way the encoder works, we need to use an EncoderParameters object; this is really just an array of parameters. You can set any number of parameters and fine-tune the way the image is saved to disk, but we are interested only in the main Quality setting. This takes values from 0 to 100, with 100 meaning full quality (no compression) and 0 meaning lowest quality (maximum compression). A compression of 50 creates a small file with reasonable quality (256x256 pixels usually takes 10KB):

```
EncoderParameters eps = new
    EncoderParameters(1);
eps.Param[0] = new EncoderParameter(
    System.Drawing.Imaging.Encoder.
        Quality, 50L);

```

With the encoder ready to do its job, we can step through each of the files selected by the user and save a compressed thumbnail:

```
Bitmap temp;
Bitmap thumbnail;
for (int i = 1; i < m_args.Length;
    i++)
{
    temp = getImage(m_args[i]);
    thumbnail = thumb(temp, m_w,

```



```

        m_h);
        thumbnail.Save(thumbdir +
            Path.GetFileName(m_args[i]),
            jpegCodec, eps);
    }
    Application.Exit();
}

```

Finally, we have to code the Cancel button's event handler:

```

private void button3_Click(
    object sender, EventArgs e)
{
    Application.Exit();
}

```

IMPROVING THE THUMBNAILS

After publishing the project and running it once to install it, you can try out the thumbnail maker. Select a few suitable JPEG pictures, right-click and select Send To, Thumber. Select the size you want the Thumbnails to be and click the Save All button. You should see that a Thumbs directory has been created complete with the thumbnails you requested.

If you look at the thumbnails produced, you might be disappointed by their quality, which is often below what you'd expect from the number of pixels; the thumbnail looks blocky. This isn't anything to do with the compression being used, as you might think. Setting the Quality to 100 or saving in an uncompressed format gives a result that's just as blocky. The solution to the problem is remarkably simple; just call `GetThumbnailImage` twice.

You can see this working if you compare the first thumbnail shown in the preview with subsequent ones as you move the slider; the first thumbnail looks blocky, but the rest look much better. So to improve the quality of the thumbnails saved to disk, simply change the single call to `thumb` within the `For` loop to:

```

temp = getImage(m_args[i]);
thumbnail = thumb(temp, m_w, m_h);
thumbnail = thumb(temp, m_w, m_h);

```

Once we have made this change, the thumbnails produced are good enough for most uses.



▲ Select the size and click Save All, and you will have thumbnails

However, if this bizarre requirement has shaken your trust in `GetThumbnailImage`, you could take control of the downsizing task more fully. You can create thumbnails using the general rescaling and resampling facilities provided in the .NET Framework to allow you to resize high-quality images with minimal degradation. This is more advanced, though, so if you are happy with easy thumbnails, skip the next section.

To use the resampling facilities, all we need is a replacement for the `thumb` function:

```

public Bitmap thumb(Bitmap image, int
    m, int n)
{

```

We need a `Bitmap` object to hold the new thumbnail we are going to create, and this must use the same `PixelFormat` as the original `Bitmap`:

```

    Bitmap temp = new Bitmap(m, n,
        image.PixelFormat);

```

It also needs to use the same resolution as the original so it is displayed at the correct size:

```

    temp.SetResolution(
        image.HorizontalResolution,
        image.VerticalResolution);

```

The idea is to draw the original `Bitmap` on to the new thumbnail `Bitmap` using graphics methods. To do this, we need a graphics object; this will allow us to draw on the thumbnail `Bitmap`:

```

Graphics g = Graphics.FromImage
    (temp);

```

The graphics object has an `InterpolationMode` property that determines how it will use the original pixels to generate a rescaled image. One of the highest-quality resampling methods is Bicubic interpolation. However, this can be slow, so you might want to experiment with the other options:

```

g.InterpolationMode = System.Drawing.
    Drawing2D.
        InterpolationMode.
            HighQualityBicubic;

```

Finally, we draw the original image at its new size and dispose of the graphics object:

```

g.DrawImage(image,
    new Rectangle(0,0,m,n),
    new Rectangle(0,0,image.Width,
        image.Height),
    GraphicsUnit.Pixel);
g.Dispose();
return temp;
}

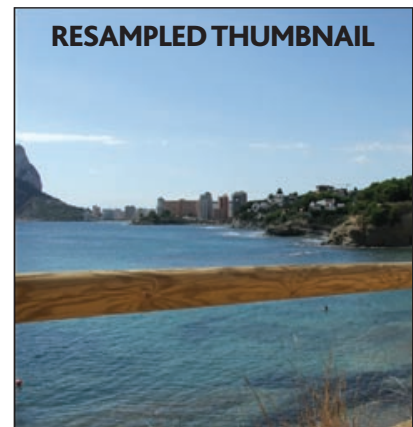
```

If you try this out, you probably won't see much difference in the resulting thumbnails until you increase the quality of the images to 80 or better.

WHERE NEXT?

There are several modifications you could make to improve the thumbnail utility. You could give the user a choice of output format (such as TIFF, GIF or JPEG) and a choice of quality. You could add another button, called Send to Messenger, to send the resulting thumbnail to a chat window. For this, all you have to do is copy the thumbnail to the clipboard, activate the Messenger application and paste the file into the chat window. You can use the Send To menu mechanism to implement any utility that processes a set of files.

Pass master Thumbnail quality using different methods



▲ Our programming uses three different methods to generate thumbnail images. Two-pass thumbnails look much better than when you call `GetThumbnailImage` once. Resampling makes for a slightly sharper result